Datalog Bottom-up
is the Trend in the Deductive Database
Evaluation Strategy

Yurek K. Hinz
INSS 690

University of Maryland, 2002

Table of contents

Abstract	p.3
Introduction	p.4
Deductive database	p.4
A brief history of logic programming	p.5
Prolog	p.5
Datalog	p.6
Deductive databases and logic programming languages	p.7
Top-down and Bottom-up evaluation strategies	p.7
Top-down technique	p.8
Advantages of Top-down	p.9
Disadvantages of Top-down	p.9
Bottom-up technique	p.10
Advantages of Bottom-up	p.11
Disadvantages of Bottom-up	p.11
Top-down and bottom-up evaluation compared	p.12
Magic sets transformation	p.12
Cost comparisons	p.13
Space optimization	p.14
Duplicate elimination	p.14
Conclusions	p.14
References	p.16

Abstract

As this work will hopefully demonstrate, the Datalog bottom-up is the trend in the deductive database evaluation strategy. Deductive databases provide more expressive power than most relational databases, because they can naturally represent non-first-normal-form relations, and allow recursive view definitions. It is easy to embed a programming interface to a deductive database in a general-purpose, or imperative, logic programming language such as Prolog. Search algorithms in deductive databases are generally divided in two parts: bottom-up algorithms and top-down algorithms. Bottom-up algorithms generate logical consequences of the database until all answers to the goal are found. Top-down algorithms start with the goal and reduce it to subgoals. When we compare a typical Prolog top-down execution with the Datalog bottom-up execution we notice that the bottom-up evaluation reduces the overall cost due to constant per-join overheads, such as initialization costs, and per iteration overheads, such as updating the various predicate extensions and increases the degree of set-orientation. Bottom-up also guarantees termination for any pure Datalog program.

Datalog Bottom-up is the Trend in the Deductive Database Evaluation Strategy

Introduction

A typical database application consists of three parts, which are usually coded in three different languages. At the top is the user interface, which manages the dialog between the user and the program. It defines menus, buttons, dialog boxes, and so on. It is either written in a special language like Tcl/Tk or HTML, or is constructed with a special editor. Underneath is a layer of code which performs data format changes, combines and aggregates data, and ensures that the integrity of the database is not violated. This code is often written in an imperative language like "C". Finally, at the bottom are the database accesses, written in SQL. The database has built-in algorithms for searching and sorting it ensures the integrity of the data and manages concurrent accesses by different users (Brass, 1994). The most important service a database management system offers is the ability to efficiently answer queries. Relational database technology provides a variety of advanced techniques for evaluating complex relational expressions, including the logic languages (Manthey, 1998).

The basic assumption in relational and deductive databases is that there are no gaps in the knowledge of the facts in our databases. The database cannot store data that contains null values or data that is indefinite. In practical situations knowledge is not precise, and there are gaps concerning facts in our databases. These gaps may be due to null values in the data, may arise when we combine several databases that lead to inconsistent theories, or may occur because information is indefinite in nature, such as in military or medical applications (Fernandez, Gryz, & Minker, 1997).

Deductive database

Deductive database can be seen as a relational database with a new query language (i.e. Datalog instead of SQL). It can also be looked at as an automated "theorem prover" which allows only special kinds of formulas (Brass, 1996). The relational database defines a number of relations called predicates "extensionally" by enumerating the mentioned tuples contained in the relation, and a logic program, which defines a number of predicates “intentionally.” Because deductive database theory is based on logic programming, it borrows heavily from the field of logic programming (Fernandez, Gryz, & Minker, 1997). It combines the benefits of representational and operational uniformity, recursion, declarative querying and efficient secondary storage access. Nonetheless, significant problems remain inherent in this synthesis. The first problem area is how to naturally represent complex values consisting of nested tuples and sets. The second problem area is how to incorporate object-oriented features into deductive framework. The third problem area is how to naturally deal with higher-order features in deductive databases (Liu, 1996).

The fact is that deductive databases provide more expressive power than most relational databases. In relational terms, they can naturally represent non-first-normal-form relations, and allow recursive view definition. It is easy to embed a programming interface to a deductive database in a general-purpose, or imperative, logic programming language such as Prolog (Kemp & Stuckey, 1991).

A brief history of logic programming

Logic programming is a programming paradigm characterized by the declarative specification in a logical style of queries and constraints over an application domain. It is contrasted to imperative programming, which is based on a procedural specification style. With logic programming, queries and constraints are expressed with a homogeneous formalism (Barták, 1998).

The beginning of logic programming can be attributed to Kowalski and Colmerauer. Kowalski formulated the procedural interpretation of Horn clauses logic (Horn is the simplest case of pure Datalog) and showed that the axiom *A if B* can be read as a procedure of a recursive programming language, where *A* is the procedure head and *B* is its body. At the same time (early 1970s) Colmerauer and his group at the University of Marseille-Aix developed a specialized theorem prover, which they used to implement natural processing systems. They called it PROLOG (for Programation et Logique or Programming in Logic). It embodied Kowalski's procedural interpretation.

An important step in the development of automated theorem proving was the invention of the resolution method by Robinson in 1969 (Robinson, 1992). The so-called "question-answering systems" were developed by Green and can be seen as predecessors of deductive databases (Green, 1969). Van Emden and Kowalski introduced the minimal model of a set of definite Horn clauses in 1976 (Emden & Kowalski, 1976). In 1979 Aho and Ullman published a paper on relational algebra with a fixpoint operator. The origins of bottom-up query evaluation are traced to that paper (Aho & Ullman, 1979). When in 1982 the Fifth Generation Project started in Japan, it gave an important momentum to the development of logic programming (Furukawa, 1992).

The natural extension of logic programming is the integration with database management, based on the use of logic programming as a query language. The resulting systems combine a declarative style for formulating queries and constraints with efficient and reliable database technology for mass-memory data storage. Several names are used to describe these systems. The term deductive database highlights the ability to use a logic programming style for expressing deductions concerning the contents of a database. Knowledge base management system highlights the ability to manage complex knowledge instead of simple data. The term expert database system highlights the ability to use expertise in a particular application domain to solve classes of problems featuring large data sets (Barták, 1998).

Prolog

Prolog is the most popular logic programming language. Prolog was at first expected to be used as a database language, but some of its features were not suited to database applications and this motivated the definition of an alternative database and logic programming language, known as Datalog.

The first Prolog interpreter was not as fast as Lisp systems but this changed in mid 1970s when David Warren and his colleagues developed an efficient implementation of Prolog. The compiler, which was almost entirely written in Prolog, translated Prolog clauses into instructions of abstract machine that is now known as Warren Abstract Machine (WAM). As an example here is a procedure for computing descendent of some man in the genealogy database.

descendent(D,A): – parent(A,D).

descendent(D,A): – parent(P,D),descendent(P,A).

The definition of this procedure is recursive: first clause for finding direct descendent concludes the recursion. The second clause uses the call to the same procedure to find indirect descendent. There are more possibilities how to define the recursive clause (i.e., how to find indirect descendent).

Datalog

Datalog is a non-procedural query language based on first-order logic that consists of a finite set of rules and a query literal. In Datalog we define two types of relations: (1) base relations - physically stored in the database and (2) derived relations - usually temporary relations that hold intermediate results. Datalog was specifically designed to be used as a database language. Syntactically, Datalog is very similar to Prolog. The evolution from Prolog to Datalog assumes going from a procedural (record-oriented) language to a nonprocedural (set-oriented) language. This transformation was parallel to the evolution of database systems from the hierarchical and network data models to the relational data model.

Another characteristic of Datalog is that each relation has a unique name and fixed arity (or number of attributes). In practice, we do not associate a “name” with an attribute, but rather its argument position. Consider the following example: (Dasiewicz, 1997)

r1(A1, A2, A3, A4)

Here *r1* is the relation name and *R1* is of arity 4. Let’s look at a relation table written in Datalog:

users(user_name, dept, password, user_id)

This query program consists of a set of rules:

up(X, Y) :– users(X, "electrical", Y, Z)

and can be read as: derive all pairs (<user_name>, <password>) for users in the electrical department. In our example “up” is a derived relation. This query could also be re-written as:

up(X,Y) :- users(X, A, Y, Z) , A = "electrical".

making them equivalent to the following relational algebra expression:

$$\prod_{user_name,\ password}(\sigma_{dept=\text{"electrical"}}(users)).$$

All of the rules are built from literals: *p(A1, A2,…,An)*, where *p* is the relation name and *Ai* are the variables or constants. Names that start with an upper case letter are variables.

The general form of a rule is as follows:

$$p(x_1, x_2, \ldots, x_n) :– q_1(x_{11}, x_{12}, \ldots, x_{m1}), –, q_k(x_{k1}, \ldots, x_{mk}), e.$$

In this form *qi* are base or derived relation names, *e* is an arithmetic predicate (any number) and each x_i appearing in *p* appears in at least one of the q_i's.
The Datalog rule can be interpreted as:

p(...) is true if q_1 *(…) and* q_2 *(…) and –* q_k *(…) and e is true.*

Because Datalog often uses the bottom-up evaluation, the termination behavior is much better than that of Prolog. Normally we can assure that termination be guaranteed for a query evaluation. This is not possible for arbitrary programs.

Termination is a major issue in databases, and large subsets of queries/programs have been defined for which termination can be guaranteed.

Besides predicates and constants, Datalog-programs contain also variables, (i.e. place holders for domain elements). In Datalog an answer to an atomic query, say *Q(X),* is a set of constants that satisfy the query (Fernandez, Gryz, & Minker, 1997).

Deductive databases and logic programming languages

There are essential differences between logic programs and deductive databases that stem from the two qualities that distinguish database management systems from other programming systems: (1) the ability to manage persistent data and (2) the ability to access large amounts of data efficiently. Deductive databases provide a declarative language for describing data. Implementations of logic programming include many non-declarative features to achieve the necessary efficiency. For example, the programmer can influence the order of computation by reordering clauses of the program, or by using non-logical control primitives. Deductive databases do not provide control primitives that can change the logical semantics of the database. Query optimization is then performed by the database itself, usually using equivalence-preserving transformations (Brass, 1996).

When compared with Prolog, deductive databases are more logical and have less control, so they are closer to a theoretically ideal logic programming language. The order of rules or body literals does not matter in a deductive database, while it is essential in Prolog. A Prolog program usually is executed by calling one main predicate, so the programmer knows in advance which arguments of a predicate are bound or free, and therefore he can order the rules and body literals in an optimal way. In deductive databases this is not possible, because the user might query any predicate with any binding pattern for the arguments. It is therefore the responsibility of the system to determine a good execution order. This matches the tradition of the database community that a database should have a powerful query optimizer. But such optimizers work well only for sets of relatively simple rules and they cannot optimize all Prolog programs (Brass, 1996).

Top-down and bottom-up evaluation strategies

When it comes to query, search algorithms in deductive databases are generally divided in two parts: bottom-up algorithms and top-down algorithms (Kemp & Stuckey, 1991). Many debate whether a logic program (i.e., Prolog or Datalog) should be processed bottom-up or top-down. The two approaches have been compared and contrasted mainly on strong (terminating)

completeness over programs without function symbols, weak (non-terminating) completeness, coverage and efficiency (Brass, 1994).

The top-down approach is goal directed, often a good feature for efficiency. However, it has been shown that strict top-down approaches cannot be strongly complete over programs without function symbols (Beeri et al., 1987). Vieille has described top-down approaches in which lemmas are used to obtain termination over programs without function symbols, and there is a completeness proof for one of his methods (Vieille, 1987).

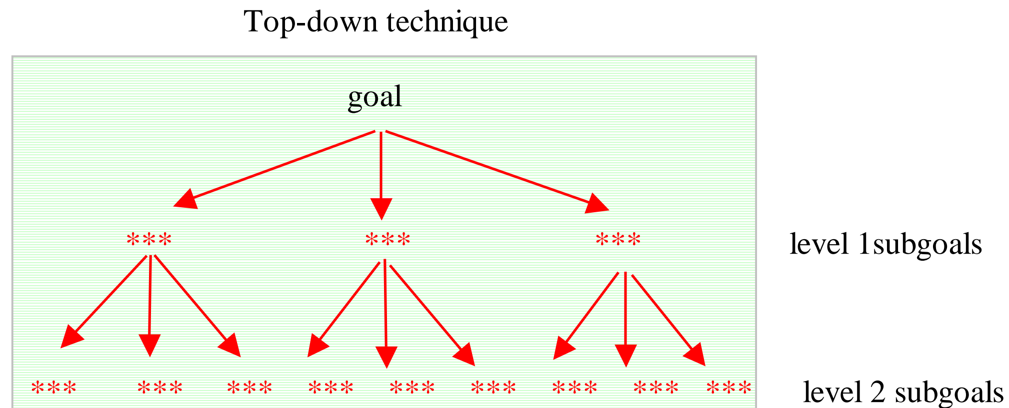

Figure:1 *Top down algorithms start with the goal and reduce it to subgoals. The subgoals are being worked on until all are solved.*

Query evaluation by top-down or backward-chaining starts with the goal which is reduced to subgoals (Muthukumar & Hermenegildo, 1991), or a number of simpler problems, until a trivial problem is reached. Then, the solution of larger problems is composed of the solutions of simpler problems until the solution of the original problem is obtained.

It is possible to simulate the bottom up computation without changes of the Prolog interpreter. The bottom-up computation starts with known facts and extends the set of known “trues” using rules. Thus, it can derive new facts from old facts and rules. This extension continues until the solved problem is present in the computed set of facts (Barták, 1998).

In general, in Prolog pure bottom-up method is less efficient than top-down method because many facts are derived that has nothing in common with the original problem. Therefore, Prolog uses top-down evaluation as a standard method of computation. However, in same cases, it is possible to guide the bottom-up evaluation towards the solution of the original problem. The main value of top-down algorithms is that top-down algorithms are goal-oriented. Thus, the search usually involves not as many irrelevant computations (Barták, 1998).

Advantages of Top-down

In certain situations Prolog's top-down execution can beat bottom-up evaluation. This example is from a paper by K. Ross (Ross, 1991). Let *P* be the program:

p(X,Z) <– ∈ (X,Y), p(Y,Z).

p(n,X) t(X).

∈(1, 2),...,∈(n-1, n).

t(1), . . ., t(m).

Query: ?-p(1,X).

Given the subgoal *?p(1,X)* Prolog sets up subgoal *?∈(1,X)* and gets an answer that binds X to 2. Using this binding Prolog sets up a subgoal *?p(2,X),* which in turn sets up subgoal *?p(3,X)* and so on until the subgoal *?p(n, X)* is set up. Prolog can deduce that there are no more answers to *∈(1,X*), and when an answer for *?p(2,X)* is found, it can directly return to the subgoal *?p(1,X)*, bypassing the subgoal *?p(2,X).* This is known as last call optimization. But applying this optimization repeatedly, when Prolog finds an answer for subgoal *?p(n,X),* it returns directly to the subgoal ?p(1;X), bypassing all intermediate subgoals. Since there are *m* answers for *?p(n,X)*, Prolog backtracks to *?p(n, X)* a total of *m* times, and evaluates the program in time $O(n + m)$. Bottom-up evaluation (using Magic Sets rewriting), on the other hand, generates each fact *p(I, j),* $1 \leq i \leq n$, $1 \leq j \leq n$, and takes $O(m * n)$ time while Prolog generates only facts *p(n,j), p(1,j),*$1 \leq j \leq m$.

Another example where top-down technique is better than the bottom-up are the cases where the user may want a single answer to a query. A top-down, tuple-oriented evaluation strategy (Prolog) sets up a query on path, and solves the subgoals. In bottom-up approach to obtain a single answer to the query is to evaluate the (magic transformed) program until we get an answer to the query, and then terminate the evaluation.

Disadvantages of Top-down

The main disadvantage of top-down algorithms in databases is that the computations are performed a tuple at a time. Reduction of a goal to subgoals involves only a small amount of data. This usually results in a loss of efficiency. Let's consider a database without built-in predicates. With the top-down evaluation method a program may not terminate.

The following example program for transitive closure iss presented in "Efficient Implementation of Loops in Bottom-Up Evaluation of Logic Queries" (Kuittinen' et al., 1994).

path(X, Y)<– edge(X, Y)
path(X, Y)<– edge(X, Z) ^ path(Z, Y).

It assumes that a directed graph is stored in the database relation edge, and computes pairs of nodes, which are connected by a path. This program runs into an infinite loop if the graph contains cycles. If the programmer knows this before, he can write a more complicated Prolog-

program, which detects cycles. Furthermore, even if there are no cycles, and Prolog terminates, it might use exponential running time (Kuittinen' et al., 1994).

Bottom-up technique

The bottom-up approach is complete and terminating in programs without function symbols. But in certain situations it computes the entire deductive closure of a program in order to answer any question, so it is not really efficient. Various methods based on “magic sets” have been proposed to focus bottom-up deduction. Sometimes a very large number of auxiliary “magic rules” is needed (Balbin, & Meenakshi, 1988 pp. 711-718). The basic logic behind magic sets is presented later in this work.

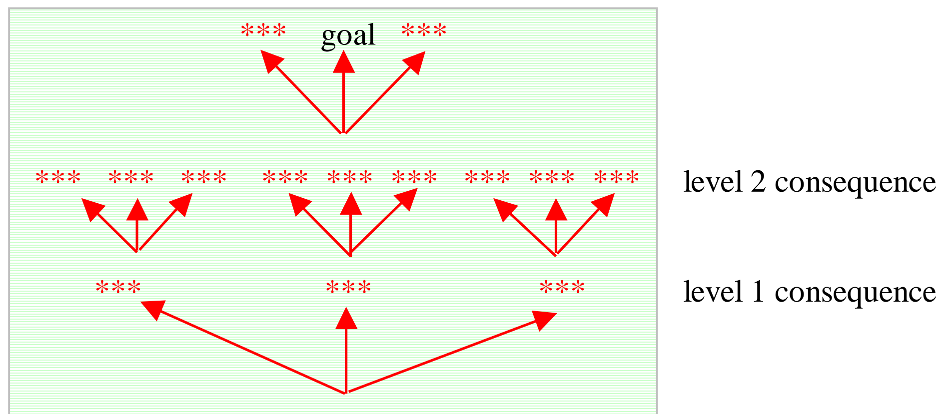

Figure:2 *Bottom-up algorithms generate logical consequences of the database until all answers to the goal are found.*

In Datalog bottom-up approach we assign rules in certain order and then evaluate them in that order. For example let's look at the relation *project(name, charges, host, user_id)*, where

nh(X,Y) :– project(X,Z,Y,U), Z > 500.
Query(N) :– nh(N, "sunee").
"return project names with more than $500 in charges"

The program has to calculate all tuples <Name, Host> with charges > $500 and select only those that are on host *"sunee."* To compare this with the equivalent relational algebra query we get:

$$\prod_{name}(\sigma_{host="sunee"}(\prod_{host,\ name}(\sigma_{charges>500}(project))))$$

The bottom-up technique allows us to optimize our original formula as:

query(N) :- project(N, Z,"sunee", U), Z > 5

making the query more efficient because now we calculate only the tuples of interest to us. This is equivalent to the relational algebra expression of:

$\prod_{name} (\sigma_{host="sunee" \wedge charges>500} (project))$

(Daszkiewicz, 1997).

Advantages of Bottom-up

Bottom-up evaluation of queries on deductive databases has many advantages over an evaluation scheme such as Prolog (top-down). In particular, it is able to avoid infinite loops by detecting repeated (possibly cyclic) subgoals. Bottom-up evaluation works by applying the rules to the given facts, thereby deriving new facts, and repeating this process with the new facts until no more facts are derivable. The query is considered only at the end, when the facts matching the query are selected (Brass, 1996).

The bottom-up method offers three significant advantages over the traditional top-down model: (1) The declarative least Herbrand model semantics is guaranteed for positive Horn clause programs; (2) Redundant derivations are avoided through memoing, leading to measurable gains in efficiency for programs in which goals or facts can be derived in many ways; (3) As a consequence of the first advantage, no operational guarantees need to be made, and a number of semantic optimizations are possible. One example of a powerful optimization made possible by the declarative semantics is factoring (Brass, 1996).

Another advantage of bottom-up evaluation of logic programs is the increased degree of set-oriented computation. When the number of inferences made by two different evaluation techniques is identical, the technique that performs more set-oriented computation is expected to perform better in terms of the number of I/O operations. Ramakrishnan's theoretical analysis and performance results show that rule ordering can greatly reduce the number of rule applications, and therefore the number of joins in bottom-up evaluation, without making additional inferences. This has two benefits: (1) It reduces the overall cost due to constant per-join overheads such as initialization costs, and per iteration overheads such as updating the various predicate extensions and (2) It increases the degree of set-orientation. The fewer the number of rule applications performed, the greater the number of inferences made in a single rule application. Again, this increases the degree of set-orientation, and hence decreases the number of I/O operations (Ramakrishnan et al., 1999). The reduction in cost due to ordering of rules depends on other techniques such as efficient join and indexing strategies, and duplicate elimination techniques.

Disadvantages of Bottom-up

The main disadvantage of bottom-up algorithms is that bottom-up algorithms are not goal-oriented. Thus, the search can involve a lot of irrelevant computation. In order to provide a goal-oriented evaluation, we must provide magic sets transformation. The code size of large programs can be considerably blown up. Debugging is made more difficult. The magic rules can be counter-productive when they contain constraints. Every materialized tuple is fitted with its own constraint store. Typically, in a recursive loop over non-ground terms, many tuples with similar arguments and constraints are created. The tuples are stored and then have to be accessed again by a search procedure (Voronkoy, 1999).

Another disadvantage to bottom-up approach is the fact that in certain situations (semi-

naive evaluation) a subsumption test for any new materialized tuple has to be performed. In such a case we are dealing with massive redundancies (Fordan, 1995). The subsumption test, which is due to the fixpoint evaluation, raises a serious efficiency problem. There are other disadvantages as well. For example, to evaluate several classes of programs with negation (or aggregation), it is necessary to order inferences. One must evaluate all answers to a negative subgoal before making an inference that depends upon the negative subgoal. Thus we have to maintain a lot of redundant subgoal dependency information (Ross, 1991).

Fortunately, the subsumption test problem can be aided by the use of modelings. Ramakrishnan, Srivastava and Sudarshan developed a model called Ordered Search model (Ramakrishnan et al., 1999). The technique is able to maintain subgoal dependency information efficiently, while being able to detect repeated subgoals, and avoid infinite loops. (Ramakrishnan et al., 1999). The Ordered Search, solves subgoals and since it performs memoing, it does not repeat computation and terminates the program. In general, it provides an alternative evaluation strategy for bottom-up evaluation. (Ross,1991).

Top-down and bottom-up evaluation compared

Many have tried to combine bottom-up and top-down query evaluation order to have both advantages: being goal directed and avoiding duplicate work. The standard method, used in many deductive database systems, is the magic set transformation (Bancilhon et al., 1986). It introduces new predicates, which encode the queries occurring during top-down evaluation, and rewrites the rules in such away that they are only applicable if the head literal is needed in order to answer the query.

To gain a better understanding of the method, consider the following example of Fibonacci numbers. To find the tenth Fibonacci number using bottom-up evaluation is virtually impossible. The bottom-up evaluation computes more and more numbers and does not terminate because the facts matching the query are selected only at the end. Top-down evaluation terminates, but it is also very inefficient. For instance to compute the tenth Fibonacci number using top-down approach, the program must evaluate 177 calls, because it does not remember which numbers it has computed already.

Magic sets transformation

As demonstrated before, the efficiency is the main problem with the bottom-up approach. To cope with the problem many techniques are developed to improve the efficiency of the bottom-up query technique. Magic sets transformation is one of the ways used to imitate top-down computations using bottom-up computation. Magic sets, is based on rewriting a logic program so that bottom-up fixpoint evaluation of the program avoids generation of irrelevant facts (Beeri, & Ramakrishnan, 1987). It is widely believed that the principal application of the magic sets technique is to restrict computation in recursive queries. The major advantage they provide is that they allow a bottom-up computation to be focused with respect to the query, thus improving the efficiency of answering queries (Kemp & Srivastava, 1991).

To gain a better understanding of the processes involved in magic set transformation let's consider our Fibonacci number problem. The magic sets transformation of bottom-up evaluation introduces a predicate, which contains the arguments, for which the Fibonacci function has to be

evaluated. In such a case the rules are only applicable if the resulting Fibonnaci number is really needed. The only stipulation is that the predicate has to be defined in such a way that it contains those arguments to the Fibonacci function which occur during the computation. Now, when this program is evaluated bottom-up it computes only Fibonacci numbers needed for the query, and computes every Fibonacci number only once. The goal-direction is inherited from top-down evaluation, while the memoing of already computed facts is from bottom-up evaluation.

Bottom-up evaluation, after the magic set transformation, is at least as efficient as top-down evaluation (Ullman, 1989). Brass argues that Ullman, who is one of the main supporters of the bottom-up evaluation approach, is mistaken. Ullman claims, that deductive databases can be equally successful to the top-down approach because of the additional functionality. But even Brass agrees that good implementations of deductive databases, using Datalog, will eventually beat Prolog systems (Brass, 1996).

There is an ongoing discussion about whether magic sets is bottom-up or a top-down procedure. The answer to this question is to a certain degree a matter of interpretation rather than of truth and error (Manthey, 1998). If we look solely at the way the rewritten rules are applied (namely by fixpoint computation), they are clearly bottom-up, and nothing else. But fixpoint iteration is just a means for materializing all derivable data. When posing a particular query, only a fraction of them will be actually needed for answering that particular request. At least in theory, complete materialization of the "virtual" part of the deductive database combined with subsequent selection of the answer facts can be taken as a universal query evaluation method (Manthey, 1998).

Current Prolog implementations typically use a form of tail recursion optimization that for certain examples, such as the right-linear version of transitive closure, will avoid rippling answer tuples up the rule/goal tree, and thus can be faster than Datalog. Although it is true that Prolog implementations have a tail recursion optimization, the main goal of this optimization is to save memory. The improvement of the running time is only a side effect and Datalog bottom-up implementation is still a better choice (Ramakrishnan et al., 2000). When we compare a typical Prolog evaluation with a bottom-up execution based on Magic Templates with Tail Recursion optimization, we notice the following: (1) Bottom-up evaluation makes no more inferences than Prolog for range-restricted programs. (2) For a restricted class of programs, which properly includes safe Datalog, the cost of bottom-up evaluation is never worse than a constant, times the cost of Prolog evaluation, and often is much better than Prolog for many programs (Ross, 1991).

Cost comparisons

Ullman compared the “time cost” of evaluation of Datalog programs using bottom-up evaluation. This comparison showed that for the class of safe Datalog programs, where only ground facts are computed and no un-interpreted function symbols are allowed, bottom-up evaluation can be performed with no more than half of the cost than that of the top-down evaluation (Ullman, 1988). Ramakrishnan and Sudarshan compared a model of Prolog evaluation that performs tail recursion optimization with a model of bottom-up evaluation and showed that with this model of bottom-up evaluation the number of inferences and the time cost are no more than that of Prolog (Ramakrishnan, 1999). Generally, there are several problems that increase the cost per inference. These problems are faced not just by bottom-up evaluation schemes, but also by top-down evaluation schemes that perform memoization of facts.

Space optimization

Space consumption by the tuple materialization is yet another problem in bottom-up execution. In contrast to the situation in the top-down execution model, we are forced to check the feasibility of the conjunction of constraint sets rather than primitive constraints. Joins of constraint relations imply the merging of constraint stores. (Fordan, 1995). Bottom-up evaluation derives new facts, but has no mechanism built in to discard facts during the evaluation. All facts are retained till the end of the evaluation. It is important to discard facts during an evaluation, once they are not needed, for otherwise space requirements may grow in an unbounded fashion. Facts are needed to make derivations and to detect duplicate derivations of facts, which may be needed for termination (Brass, 1996). But disk space is inexpensive and usually immaterial in most cases.

Duplicate elimination

When a fact is derived in a bottom-up evaluation, duplicate checking needs to be done to see if the fact was derived earlier. If it was not, derivations can be made using the fact. If it was, the fact can be discarded. This check is important for termination of many programs. However, for some programs (such as our Fibonacci program) duplicate facts are never generated. Hence duplicate elimination doesn't need to be performed for such programs. (Maher & Ramakrishnan, 1990).

Conclusions

It is fairly easy to find examples where the behavior of Prolog top-down approach is much worse than that of Datalog bottom-up evaluation. Some programs when using top-down evaluation method Prolog does not even terminate, which is not the case with the bottom-up evaluation. Often bottom-up methods are better than Prolog over a wide range of programs, but they can do considerably worse for some programs that manipulate large non-ground terms. Bottom-up methods also have the virtue of being complete and permit the use of a wide range of additional optimizations, while sharing many of the advantages of top-down evaluation.

Among query processing algorithms in deductive database systems bottom-up evaluation is becoming the most popular, as evidenced by the survey of Bancilhon and Ramakrishnan (Bancilhon & Ramakrishnan, 1988) and research done by Ullman (Ullman, 1988 & 1989). Most deductive database systems in use today are based on bottom-up evaluation. This is very different from top-down information architecture, which involves reducing the ambiguity inherent in unformulated situations (Rosenfeld, 1998). There are many optimization models used to improve the efficiency of the bottom-up approach (i.e., magic sets) but they go beyond the scope of this work. Datalog bottom-up queries are able to avoid infinite loops by detecting repeated subgoals. Further, in many database applications, it is more efficient than Prolog due to its set-orientedness (Ramakrishnan, Srivastava, & Sudarshan, 1999). Because in Datalog bottom-up model the declarative semantics is guaranteed for positive, no operational guarantees need to be made. Redundant derivations are avoided through memoing, leading to measurable gains in efficiency. The reduction in cost can be achieved through ordering of rules and other techniques such as efficient join and indexing strategies, and duplicate elimination techniques. Thanks to

the benefits of the new models (such as magic sets) the bottom-up information architecture is getting better.

The challenge of bottom-up architecture is to reduce the ambiguity inherent in large, imprecise bodies of content. Bottom-up architecture helps to embrace the content of database and makes organizing and managing it much more capable then before. This in turn should help users find what they're looking for in these huge amounts of content. Combined with technologies like XML bottom-up will continue to be a trend in deductive database query (Rosenfeld, 1998).

References

Aho, A., & Ullman, J. (1979). *Universality of data retrieval languages.* ACM Symposium on Principles of Programming Languages, 110-120.

Balbin, I., Meenakshi, K., A. (1988). *Query Independent Method for Magic Set Computation on Stratified Databases.* International Conference on Fifth Generation Computer Systems, Tokyo, Japan, pp. 711-718.

Bancilhon, F., Maier, D., Sagiv, Y,. & Ullman, J. (1986). Magic sets and other strange ways to implement logic programs. *ACM Press*.

Bancilhon, F., & Ramakrishnan, R. (1988). An Amateur's Introduction to Recursive Query Processing Strategies, *Readings in Database Systems*.

Barták, R. (1998). *Guide to Prolog Programming.* [On-Line]. Available: http://ktiml.mff.cuni.cz/~bartak/prolog/intro.html

Beckstein, C., & Kim, M. (1989). *A Mixed Top-Down and Bottom-Up Deduction Method and its Correctness*. IBM Research Division, Yorktown Heights, NY.

Beeri, C., Naqvi, S., Ramakrishnan, R., Shmueli, C., & Tsur, S. (1987). *Sets and negation in a logic database language.* In Proceedings of the ACM Symposium on Principles of Database Systems, San Diego, California, CA.

Beeri, C., & Ramakrishnan, R. (1987). *On the Power of Magic*. PODS 1987: pp. 269-283

Brass, S. (1996). *Bottom-Up Query Evaluation in Extended Deductive Databases*. Dem Habilitation dissertation for Fachbereich Mathematik der Universitat at Hannover

Dasiewicz, P. (1997). *Materials for ECE456,* Department of Electrical and Computer Engineering, University of Waterloo, Waterloo, CDN.

Emden, M., & Kowalski, R. (1976). The semantics of predicate logic as a programming language. *Journal of the Association for Computing Machinery,* 23, 733-742.

Fernandez, J., A., Gryz, J., & Minker, J. (1997). *Disjunctive deductive databases: semantics, updates, and architecture.* Institute for Advanced Computer Studies University of Maryland, College Park, MD.

Fordan, A. (1995). Linear Constraint Projection in Top-Down and Bottom-Up *.GMD-FIRST* German National Research Center For Information Technology.

Green, C. (1969). Theorem-proving by resolution as a basis for question-answering systems. *Machine Intelligence, 4*, 183-205.

Furukawa, K. (1992). Logic programming as the integrator of the fifth generation computer systems project. *Communications of the ACM, 35 (3),* 82-92.

Kemp, D., Stuckey, P. (1991). *Magic Sets and Bottom-up Evaluation of Well-Founded Models.* (Paper) Bull Information Systems, Phoenix, AZ.

Kuittinen , J., Nurmil, O., Sippu, S., & Soisalon-Soininen, E. (1994). *Efficient Implementation of Loops in Bottom-Up Evaluation of Logic Queries.* Department of Computer Science, University of Helsinki, SF.

Liu, M. (1996). *Deductive Databases Where to Now?* (Paper). Department of Computer Science University of Regina, Regina Saksatchewan, CDN.

Maher, M., & Ramakrishnan, R. (1990). *Deja vu in fixpoints of logic programs.* In Proceedings of the Symposium on Logic Programming, Cleveland, OH.

Manthey, R. (1998). Datalog and beyond: A "Gentle" Introduction to Research in Deductive Databases, University of Bonn, D.

Muthukumar, K., & Hermenegildo, M. (1991) Combined determination of sharing and freeness of program variables through abstract interpretation, pp. 49--63, Paris, F.

Ramakrishnan, R. (1998). *A Survey of Research on Deductive Database Systems.* University of Wisconsin, Madison, WI.

Ramakrishnan, R., Srivastava, D., & Sudarshan, S. (1999). *Controlling the Search in Bottom-Up Evaluation,* University of Wisconsin, Madison, WI.

Ramakrishnan, R., Srivastava, D., & Sudarshan, S. (2000). *Top-Down vs. Bottom-Up Revisited.* University of Wisconsin, Madison, WI.

Robinson, J. (1992). Logic and logic programming.

Communications of the ACM, 35 (3), 41-65.

Rosenfeld, L. (1998). *Bottom-Up Architectuer.* [On-Line]. Available:

http://www.webreview.com/1998/08_14/designers/08_14_98_2.shtml

Ross, K. (1991). *Modular Stratification and Magic Sets for Datalog programs with negation.* ACM Symposium on the Principles of Database Systems.

Ullman, J. (1988). Principles of Database and Knowledge-Base Systems, *Computer Science Press, (1).*

Ullman, J. (1989). *Bottom-up beats top-down for datalog*. In Proceedings of the Eighth ACM

Symposium on Principles of Database Systems, pp. 140-149, Philadelphia, PA.

Vieille, L. (1987). A database-complete proof procedure based on SLD-resolution. *International Conference on Logic Programming, Melbourne*, Australia, 74-103.

Voronkoy, A. (1999). *Deductive Database*. Computing Science Department Uppsala University Uppsala, S. [On-Line]. Available: http://www.csd.uu.se/~voronkov/ddb.htm